Improving Text-to-Code Models with Diverse Data Augmentation Techniques

ZDKP6[1]

MSc.Data science Machine learning

Supervised by

Dr. Efstathia Christopoulou
Prof. Matt Kusner

Submission date: September 2023

Acknowledgments

Firstly, I would like to thank my industrial supervisor, Efstathia (Fenia) Christopoulou, from Noah's Ark Lab, for her kind supervision and patient guidance throughout the entire experiment. Your support has been invaluable.

Secondly, I would like to thank to my parents and grandmother for their care and support over the years. I am grateful for everything you have done for me, and credit for whatever I have achieved goes to them; the mistakes are only mine.

I acknowledge the use of ChatGPT-4 (Open AI, https://chat.openai.com) to proofread my final draft.

Abstract

The growing demand for code-oriented tasks and the lack of datasets for text-to-code generation has emphasized the importance of data augmentation for text-to-code tasks. This dissertation investigates the impact of various data augmentation methods on the text component of text-to-code tasks to improve code model performance. It includes an analysis of the performance of three groups of augmentation methods: word replacement, noise addition and paraphrasing. Our results show that synonym replacement methods may replace key components of natural language description of programs, leading to poor performance, while noise addition methods offer some model robustness. We find that paraphrasing methods such as back-translation, are the most effective.

In our experiments, we obtained a 0.85% improvement over the baseline when doubling the size of the training data via back-translation. We then investigated possible improvements by using less training data. We thus compared the performance of the model using half of the original data, with and without augmentation, to its performance when using the full dataset. The results highlighted the fact that additional data is not always beneficial and possibly smaller but higher quality data are more important. By plotting the learning curve for various augmented data sizes, we found that an optimal selection of back-translated examples (75%) resulted in a 1.13% higher CodeBLEU score compared to the baseline.

Source code for this thesis:

`https://github.com/andrew71938/MasterProject-DA-text2code`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview and Significance

The rapid advancement of computer science in recent years has led to a significant demand for software developers. A study by Evans Data Corporation indicated that there were approximately 23.9 million professional software developers in 2019, and this number is expected to increase to 28.7 million by 2024 (Lu et al. 2021). This notable increase highlights the importance of language models for code, as they can help streamline tasks such as code debugging and basic code synthesis. Products like ChatGPT [1] and Copilot [2] have received significant attention recently, demonstrating the growing interest in this area. A crucial aspect of this development is the task of text-to-code generation, which involves converting natural language (NL) into computer programming language (PL). Successfully completing this task can help new software developers and those with limited programming experience to adapt and use programming tools more efficiently.

Text-to-code generation is often seen as a sequence-to-sequence task, similar to language modeling and machine translation. It involves converting NL into computer code. While this is fundamentally a sequence-to-sequence task (Sutskever et al. 2014), it is distinct from other similar challenges due to the need for accurate outputs. This is because PLs have strict syntactical rules and complex logical patterns, as well as unique tree structures in many cases. As a result, general models designed for regular text translations may not be suitable for generating precise code, highlighting the need to pre-train models using code sources and then fine-tune them using text-to-code benchmark datasets.

However, the datasets for the text-to-code task are limited compared to text-to-text tasks such as machine translation, which contain several millions of training datasets, whereas

[1] https://openai.com/chatgpt
[2] https://github.com/features/copilot

the dataset for the text-to-code task is only around several thousands (Zhu et al. 2022). Hence, to improve the model's performance and make the model more adaptable to noise, we experiment with several data augmentation methods. Augmentation methods are usually

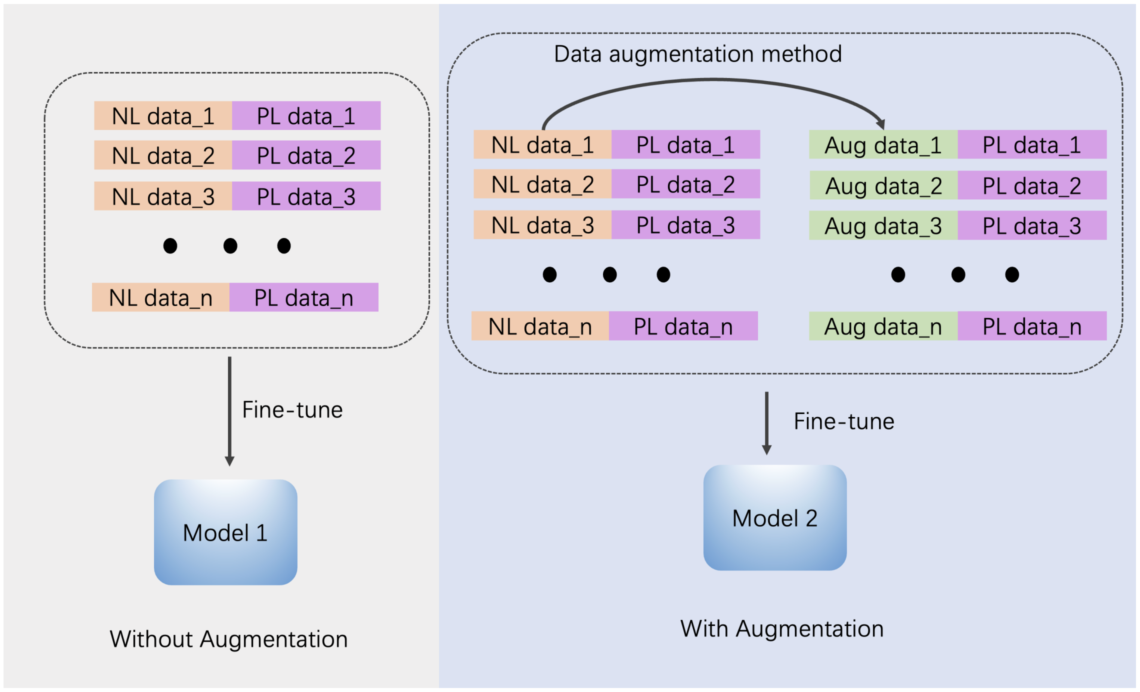

Figure 1.1: Procedure of data augmentation

utilized in Natural Language Processing (NLP) to improve model performance without the need for extra datasets. In this thesis, we apply data augmentation techniques on the natural language description part of text-to-code tasks. Consequently, we combine the original data and the augmented data while fine-tuning Code Language Models on the combined data. This approach aims to improve the model's performance in code generation, as illustrated in Figure 1.1, and become a solution to the dataset shortage problem for text-to-code tasks. This method also can make the models more robust by adding noise to the training dataset.

1.2 Research Questions and Main Challenges

1.2.1 Research Questions

1. Given that several data augmentation methods exist, some of which are not commonly applied on sequence-to-sequence tasks and have unknown performance characteristics, it is critical to assess various methods to determine the most effective one for the task of text-to-code generation. Such an evaluation involves the application of multiple aug-

mentation techniques, comparing their performance on the XLCoST benchmark (Zhu et al. 2022), and determining which one that has the most significant improvement over a model that has been trained on non-augmented data. In summary, the question is: **"Which data augmentation method is the most effective and why, and why other methods are not as effective in the task of text-to-code generation?"**

2. After identifying the most effective data augmentation method from our evaluation, it is essential to delve deeper into its performance characteristics. This involves a detailed analysis of the method's impact on the model's performance, checking how large is the improvement compared to the baseline, and assessing if there is potential for further enhancements. Hence, the question can be: **"Does the selected data augmentation method result in a satisfactory improvement of the model's performance? Is there room for further enhancements of this method?"**

1.2.2 Main Challenges

1. The chosen dataset for this experiment must be sufficiently large to allow modifications by the data augmentation methods in NL and should also be readable for analysis. Furthermore, the selected model must have certain parameters that enable accurate results for code generation. Some models may not be sophisticated enough to generate long outputs, and the selection process must also consider computational resources. Hence the first main challenge is: **"Selecting the appropriate dataset and model."**

2. Comparing the performance of each augmentation method with a baseline model is the first step. To further understand and potentially improve each method, it is necessary to inspect the augmented text and compare it with the original text. This analysis will help in understanding why a particular method causes a positive or negative impact on the resultant model. This task is challenging because it requires combining the intrinsic ideas of data augmentation methods with practical examples. In conclusion, this challenge could be address as: **"Understand and analyze the augmentation methods by practical examples."**

1.3 Structure of the Thesis

The thesis is structured into six main chapters, each designed to provide a comprehensive understanding of the applying data augmentation on text-to-code generation tasks.

In Chapter 2, we lay the foundational background, beginning with an exploration of the transformer architecture, followed by an examination of the evolution of pre-trained models in catering to code-oriented tasks. Additionally, we focus on the evaluation metrics designed specifically for code-related tasks.

In Chapter 3, we provide a comprehensive review of the text-to-code task, delving into both foundational concepts and recent developments in the field. This is followed by a detailed explanation of data augmentation techniques and concludes with a discussion on related work in the domain of data augmentation for code tasks.

In Chapter 4, we categorize various augmentation methods we used in this thesis into three primary categories: word replacement, noise addition, and paraphrasing. We first describe these categories and the techniques they involve. We then apply these augmentation techniques on our selected dataset and fine-tune a selected model on the original and augmented data. We then present our results and delve deeper into the analysis of each data augmentation method by selecting a working example. Finally, we draw an overview and highlight the limitations of our study.

In Chapter 5, we experiment with by fine-tuning on less data and find that additional training data is not always associated with better performance. This is followed by further improvements obtained by applying back-translation on those data .

Chapter 6 provides a summary of the thesis experiments, a self-evaluation of the outcomes, and a discussion on future work.

1.4 Our Contribution

Comparing a wide range of text-to-text tasks, such as machine translation, reveals that code-oriented tasks form a smaller section. Within this section, text-to-code is a subpart of code-oriented tasks, where most work focuses on code-to-code translation, and only a few studies involve text-to-code. This study aims to evaluate the impact of various data augmentation techniques on text-to-code tasks, with the ultimate goal of maximizing improvement from data augmentation. We employed a range of data augmentation techniques, categorized into three groups: word replacement, noise addition, and paraphrasing, to enhance the model's performance. Notably, the paraphrasing methods, specifically the Back-translation approach (Berard et al. 2019), yielded the highest CodeBLEU score of 38.84% using German as the pivot language (Ren et al. 2020).

However, while some techniques like paraphrasing and noise addition (word swap and deletion) showed promise when selecting appropriate hyper-parameters, others like word replacement and TF-IDF consistently under-performed or hovered around the baseline. During the synonym replacement method, we also employed pre-trained models, such as BERT and Word2Vec to replace words (Devlin et al. 2018; Mikolov et al. 2013). The results were not ideal because these models tend to modify the key order and figures. Large language models can avoid the problem of replacing key figures and orders, as shown in the use of GPT-3 for paraphrasing methods(Brown et al. 2020b). The results of this method perform well even in back-translation.

Despite this, results show that the highest improvement compared to the baseline was approximately 0.85%. Although the improvement is minimal, we conducted further analysis to determine whether the performance using half of the data is close to the performance of models fine-tuned using the full dataset. This analysis using only half of the data resulted in a 1.7% difference compared to using the full data. During the experimentation, we realized the importance of adjusting the scale of back-translation, which we then applied to the full dataset to optimize performance. Specifically, we discovered that applying back-translation to 75% of the dataset resulted in a peak performance, yielding a 39.12% CodeBLEU score (Ren et al. 2020). This score is 0.28% higher than the best performance score previously achieved and 1.13% higher than the baseline. We also showed how the scale in back-translation affects the results.

These contributions provide useful information regarding data augmentation techniques for text-to-code generation tasks, and a detailed analysis of each augmentation method's performance by analyzing practical samples. Our findings indicate that more back-translated data is not necessarily better. This is an important area that necessitates further research related to the quality of back-translated data and its scale for text-to-code generation.

Chapter 2

Background

In this chapter, we will present foundational background information on data augmentation within the realm of text-to-code tasks. Our discussion begins with an exploration of the transformer architecture, followed by an examination of how pre-trained models have evolved to cater to code-oriented tasks. Additionally, we will focus on the progression of evaluation metrics tailored specifically for code-related applications.

2.1 The Transformer Architecture and its Evolution in NLP

Transformer is a seminal architecture in NLP that stands distinct from prior models like CNNs (Krizhevsky et al. 2012), LSTMs (Hochreiter and Schmidhuber 1997), and RNNs (Sutskever et al. 2011). It introduces the self-attention mechanism (Vaswani et al. 2017), which enables the model to weigh the significance of each word in a sentence when predicting the next word. This mechanism is also parallelizable, which makes it more efficient than LSTMs.

In subsequent sections, we will focus on the function of the attention mechanism, the encoder-decoder structure within the Transformer framework and several NLP models based on this structure.

2.1.1 Self-Attention and Multi-head Attention

The attention mechanism is an integral component of the Transformer architecture (Vaswani et al. 2017). In this mechanism, it's essential to understand its three primary components: query, key, and value. The query can be conceptualized as the word or phrase for which we seek context, acts as asking a question. The value functions as a database, containing potential contextual answers or representations for any given input. The key serves as a label, facilitating the matching of the correct value in response to a particular query. Through this mechanism, the attention process identifies and assigns higher significance to the most contextually relevant values for a given query.

Self-Attention

Self-attention aims to capture the inter-dependencies within a sequence, regardless of their distance from one another. The mechanism achieves this by computing attention scores using Query (Q) and Key (K) matrices first. Specifically, the dot product of Q and K is computed to represent how much each element in the sequence should attend to every other element. This approach is based on the premise that aligning the query with the key provides a suitable metric for inter-element relevance (Vaswani et al. 2017).

However, the raw scores resulting from the matrix multiplication between Q and K can be large and varied. To ensure these scores are comparable and to accentuate the differences among them, we employ the softmax function:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.1}$$

At this stage, a potential challenge with the softmax function is the vanishing gradient problem, particularly when the raw scores are very large or small. This can influence effective learning during the back-propagation process, as the gradients can be exceedingly minute.

To solve this issue, a normalization step is introduced before applying the softmax function (Britz et al. 2017) . Specifically, the raw attention scores are divided by the square root of the dimension of the Key vectors, $\sqrt{d_k}$ This normalization prevents the softmax function from squashing the scores too severely into the extreme ends of the output range.

The output of the softmax function provides normalized attention weights, which signify the importance of each element in relation to others. Multiplying these weights by the Value (V) matrix then gives us the self-attention output, hence to repersenting the

$$\text{Attention} = Softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

which effectively captures the weighted context of each element based on the entire sequence. Thereby enhancing the model's ability to find dependencies irrespective of their positional distance.

Multi-head Attention

Multi-head attention, as its name suggests, integrates multiple self-attention mechanisms working in parallel. It employs multiple independent sets of projection matrices, thus allowing the model to focus on different parts of the input simultaneously. These individual attention outputs are then linearly combined, as linear combination aids in preserving and using information from all heads effectively (Vaswani et al. 2017).

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W^O \tag{2.3}$$

In the above equation, W^O represents the output projection matrices, which linearly transform the concatenated attention heads into the desired dimension.

The purpose of this composition is not merely to increase model parameters, but to allow the model to capture diverse representations of the data. By employing a linear concatenation, each head can preserve its unique learned perspective, ensuring that specific features or relationships, which might be predominant in one head over the others, are retained.

Consider the sentence: “Apple is a fruit that contains vitamin C.” In processing this sentence, one attention head might focus on the relationship between “apple” and “fruit,” understanding the categorization. Another head might capture the attribute relationship between “apple” and “vitamin C,” emphasizing the nutritional aspect.

In conclusion, multi-head attention equips the transformer architecture with the ability to simultaneously process and represent various facets of information, thereby enhancing its capacity to understand complex inter-relations within the data.

2.1.2 Basic Construction of Transformer

Both the self-attention mechanism and multi-head attention are important component in transformer architecture. To comprehend their performance, it’s crucial to understand the foundational structure of the transformer, which is constructed based on the encoder-decoder paradigm. This architecture is illustrated in the figure below.

As display in Figure 2.1, the encoder comprises multiple layers. Each layer, as described in the original paper, consists of sub-layers, one of which is the fully connected feed-forward network. The feed-forward function operates on the principle of positional encoding. Positional encoding is imperative because, in the absence of recurrence or convolution in the model, it facilitates the understanding of sequence order, embedding information regarding the relative or absolute token positions in the sequence. Hence, ensuring the sequence’s

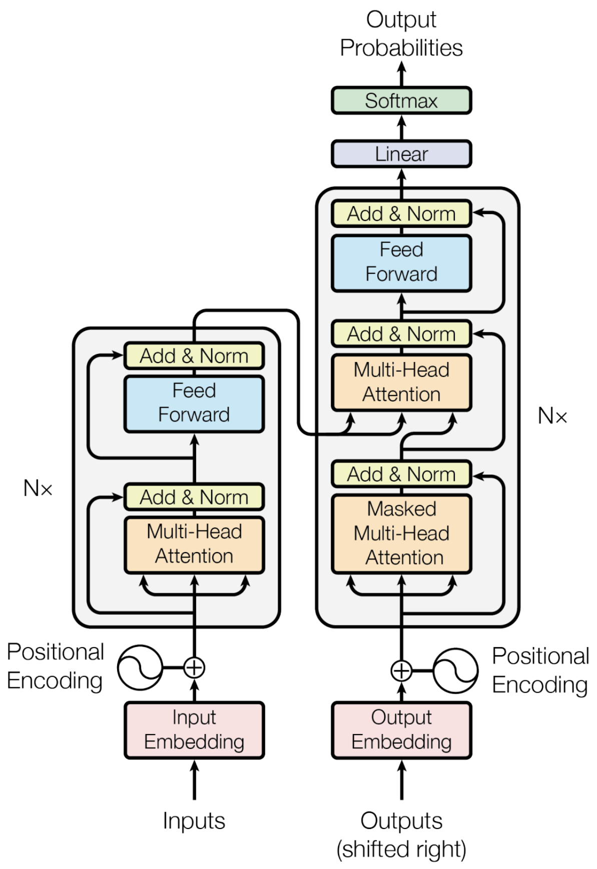

Figure 2.1: Architecture of the transformer (Vaswani et al. 2017)

temporal or ordinal essence is captured.

$$\text{Residual}(x) = x + F(x) \tag{2.4}$$

Around each of the sub-layers, a residual connection is employed, which is then followed by layer normalization.

$$\text{LayerNorm}(x) = \gamma * \left(\frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\right) + \beta \tag{2.5}$$

Layer normalization stabilizes and accelerates deep neural network training by normalizing input across features, contrasting with the traditional approach of normalizing across batches. In the equation presented, x symbolizes the input, while γ and β are trainable scaling and shifting parameters, respectively. This normalization technique addresses the challenge of internal covariate shift, ensuring a smoother training process. Concurrently, residual connections tackle the vanishing gradient problem, facilitating the training of deeper architectures. These connections craft a "shortcut" pathway, streamlining the flow of gradients during back-propagation.

The decoder, while sharing similarities with the encoder, possesses distinct attributes. Apart from the two sub-layers found in each encoder layer, the decoder introduces a third sub-layer for multi-head attention over the encoder's output. This structure maintains the residual connections and layer normalization seen in the encoder. Also, due to linguistic patterns, the self-attention mechanism within the decoder is modified. This modification restricts tokens from attending to future tokens. Such masking, in connection with the offsetting of output embeddings by one position, guarantees that predictions for a given position i are solely based on known outputs from preceding tokens.

Masking plays a pivotal role as it ensures that during training, the model doesn't get access to future tokens, thus aligning with the sequential nature of language where future words are unknown during real-time processing. This makes the temporal order of language is essential for tasks such as translation, where predicting a word relies heavily on preceding words in the source text.

2.1.3 Transformer-based Models

Models constructed using the transformer architecture can generally be categorized into three structural configurations: encoder-decoder, encoder-only, and decoder-only. In this section, we will explore three prominent models, each exemplifying one of these configurations: T5 represents the encoder-decoder paradigm, BERT exemplifies the encoder-only structure, while GPT is a notable instance of the decoder-only design.

BERT

BERT, which stands for Bidirectional Encoder Representations from Transformers, is designed to create deep bidirectional representations by paying attention to both left and right context across all layers (Devlin et al. 2018).

BERT uses a special encoder-only design, which means BERT does not have a decoder. This design decision is based on its use of the masked language model (MLM) training method. In this approach, the objective is to predict masked words, a process that naturally requires a comprehensive understanding of the entire input sequence. Which is different from decoders that only rely on previous words. Additionally, BERT's ability to determine the relationships between sentences is improved through another special pre-training task, next to MLM, which is assessing the contextual coherence between consecutive sentences, also known as next sentence prediction (NSP).

Turning our attention to BERT's training strategy, it's mainly two-step. First, it engages in semi-supervised training on big datasets, such as literary content and Wikipedia, to develop a basic understanding of language. Subsequently, supervised fine-tuning sharpens BERT's skills, making it effective in different contexts and adaptable to specific applications like sentiment analysis such as spam classification.

A key benefit is that after pre-training, BERT can be easily fine-tuned with small changes to its structure, leading to top performance across many tasks. Also, during pre-training, BERT's occasional word substitution strategy improves its adaptability during fine-tuning.

Finally, BERT emerges as a bidirectional paradigm, pre-trained on voluminous data, and fine-tuned to specific tasks, thereby establishing benchmarks in the domain of NLP.

GPT

The architecture of Generative pre-trained Transformers (GPT) markedly differs from that of BERT. Mainlt, GPT only utilizes a decoder-based structure and doesn't have any encoder elements. This decision is because GPT is a generative model, designed to create coherent and relevant text sequences. Decoders are great at this because of their autoregressive nature. In tasks like conversational AI, this allows GPT to sequentially produce human-like responses, making it particularly adept for chatbot implementations. On the other hand, BERT (Devlin et al. 2018), with its bidirectional understanding of context through encoders, excels in tasks that need deep understanding of text.

GPT models are trained using unsupervised learning on large amount of text data. Initially, the model is pre-trained to predict the subsequent word in a sequence, thus enabling it to generate text. After that, it can be fine-tuned for specific tasks (Radford et al. 2018).

OpenAI has developed several iterations of the GPT model. GPT-1 introduced the foundational architecture and illustrated the potential of transformer-based generative models. It was trained on books and demonstrated impressive text generation capabilities for its time (Radford et al. 2018). GPT-2, in contrast, increased the number of parameters to 1.5 billion, leading to significant improvements in coherence and text generation quality (Radford et al. 2019). In the third iteration, GPT-3, the number of parameters reached 175 billion. This huge number of parameters made the model show emergent behavior, as it not only enhanced text generation quality but also exhibited the ability to execute tasks without the need for task-specific training data (Brown et al. 2020b). This made GPT-3 a few-shot learner, meaning it could understand tasks with just a few examples and show logical reasoning.

In conclusion, the GPT model's decoder structure, which learns by predicting the next word in a sequence, and its iterative development from GPT-1 to GPT-3, underscore the rapid advancements in transformer architectures.

T5

In NLP, there are several tasks exists, such as language translation, question answering, and classification. Historically, dedicated models were designed and trained for each task, seeking optimal performance tailored to individual requirements. This changed with the introduction of the T5 model by Raffel et al. (2020), which proposed for a paradigm shift by leveraging transfer learning.

Transfer learning, a technique where a model is pre-trained on a data-rich task before being fine-tuned on downstream tasks, has emerged as an influential method in NLP (Oquab et al. 2014). This approach's efficacy has given rise to a diversity of techniques, methodologies, and practices. Accordingly, a unified framework was proposed, aiming to harness the power of transfer learning in NLP by transforming all text-based language problems into a text-to-text format. This formation paved the way for the inception of the "Text-to-Text Transfer Transformer", which known as T5.

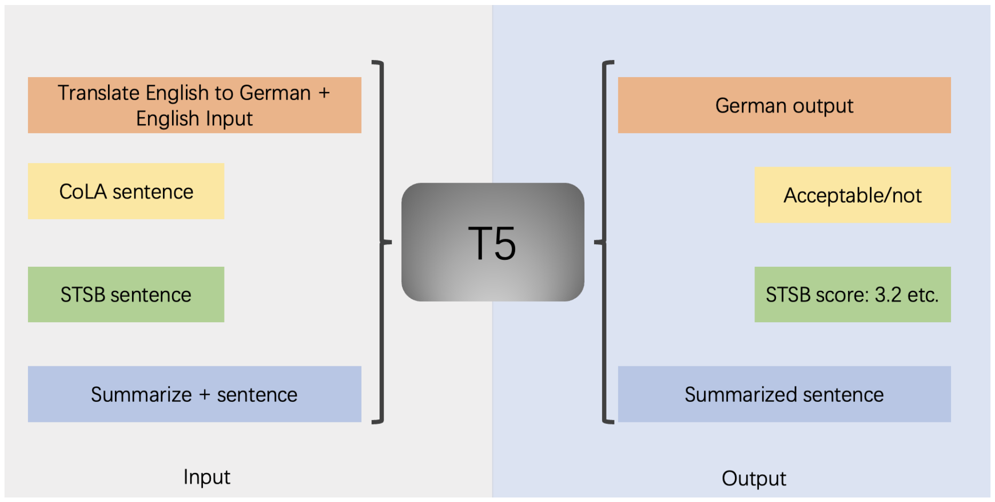

Figure 2.2: T5 model's versatility across various tasks (Raffel et al. 2020)

A distinguishing feature of T5 is its ability of a unified Text-to-Text transformation. Be it classification, regression, sequence generation, or any other NLP task, T5 standardizes it as a textual input-output mechanism (Raffel et al. 2020). This standardization inherently means that both the input and output are represented as text, simplifying many preprocessing and task-specific adaptations typically seen in NLP.

Within the above figure, tasks like CoLA (Corpus of Linguistic Acceptability) focus on the grammatical acceptability of sentences, while STSB (Semantic Textual Similarity Benchmark) evaluates the similarity of pairs of sentences. Furthermore, owing to T5's transformation to the text-to-text format, tasks are usually prefixed to signal their nature.

While combining different tasks into a text-to-text approach may seem easy in theory, it actually involves complex challenges in practice. One of the key advantages of T5 is its ability to smoothly switch from one task to another, all within a single format. Besides this, T5 includes changes from the traditional Transformer, particularly during its pre-training and fine-tuning stages. Its pre-training is not just based on language models, but includes various tasks like text summarization and translation, promoting a rich transfer learning environment. The comprehensive text-to-text structure also provides a user-friendly interface during both pre-training and fine-tuning phases.

A key factor in T5's success relates to its extensive pre-training, augmented by a massive database. By merging research insights, scalability, and their extensive new corpus, T5 has established groundbreaking benchmarks across a range of tasks, from summarization and question answering to text classification.

2.1.4 Adapting Transformers to Code

Most pre-trained model are trained by a dataset sourced from Wikipedia and other NL text, which means that although it has a strong transformer structure, it is not able to produce reasonable answers for tasks oriented towards code. Hence, some specific code models based on the structure of previous pre-trained models, with several improvements based on PL structures.

CodeBERT

CodeBERT is the first pre-trained language model tailored for tasks such as NL to code conversion. The dataset employed for its pre-training closely mirrors that of CodeT5 (Wang et al. 2021), as discussed previously. The dataset encompasses both unimodal PL data and bimodal text-to-code data, spanning six PL. Specifically, the bimodal dataset originates from Husain et al. (2019), while the unimodal dataset is sourced from GitHub repositories. Given its multi-language pre-training, CodeBERT showcases superior performance across various PL.

Structurally, CodeBERT is built based on the architecture of BERT (Devlin et al. 2018), which is an all-encoder design, ensuring bidirectional context capture. Furthermore, it has an identical model structure to RoBERTa-base, which contains 125M parameters (Liu et al. 2019). Notably, while RoBERTa is a derivative of BERT, it introduces modifications in hyperparameters during the training phase and redefines sentence segmentation. Unlike BERT, which considers sentence pairs, RoBERTa processes continuous sentences, disregarding their pair-based coherency. Additionally, RoBERTa eliminates the NSP task, centering its focus on the MLM task.

Apart from the difference in the training datasets of CodeBERT and RoBERTa, another difference is the training method of CodeBERT. It contains a paraphrasing objective—specifically, a hybrid function that integrates the replaced token detection task. This particular task is aimed at identifying possible alternative tokens generated by models, which is a part of the paraphrasing process (Feng et al. 2020). By these techniques, CodeBERT outperforms RoBERTa's performance based on same structure, setting state-of-art performances of its time for natural language code searches and code documentation generation tasks.

CodeT5

Building on the foundation laid by the T5 model, which was specifically designed for NL tasks (Raffel et al. 2020), there emerged a need to adapt and refine this model for tasks associated with PL. To this end, Wang et al. (2021) introduced CodeT5, model based on the T5 architecture. This model demonstrates exceptional performance across a range of code-related tasks.

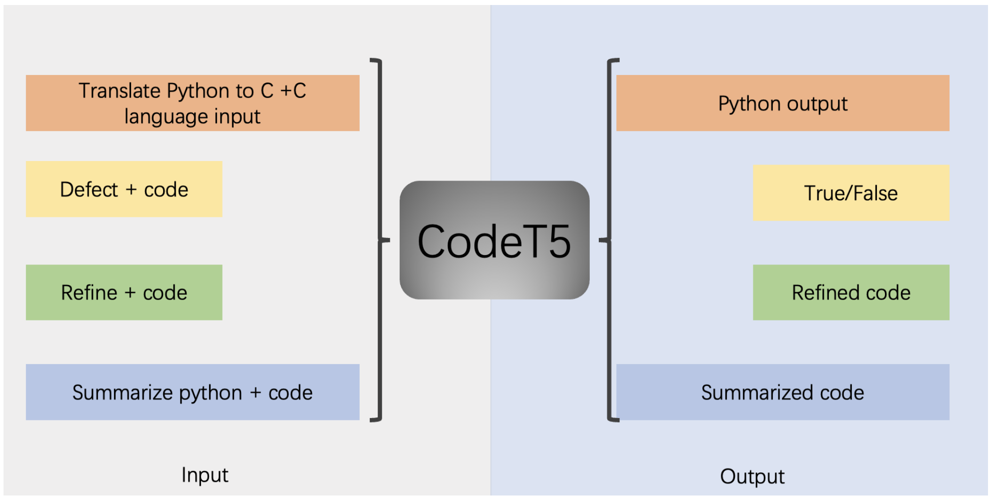

Figure 2.3: CodeT5 model's versatility across various tasks (Wang et al. 2021)

While the original T5 model exploited transfer learning to seamlessly transition multiple tasks into a text-to-text task (Raffel et al. 2020), the shared architecture ensures that CodeT5 retains similar capabilities in transfer learning. For instance, it excels in tasks such as code summarization, code translation, and code refinement, as shown in Figure 2.3. Incorporating a prompt at the outset enables the model to learn the intricacies of a task and generate precise results. The encoder-decoder structure ensures that while the encoder comprehends the code, the decoder is primed to produce a relevant output. The importance of next-token prediction is evident in PL, which require strict following of structure, especially in tasks like code generation.

Diverging from the original model, CodeT5 different compared to the original T5 structure mainly in two ways:

1. Pre-training Data: Unlike its predecessor, CodeT5 is tailored using the CodeSearchNet dataset (Husain et al. 2019), as described by Feng et al. (2020). This dataset comprises both unimodal PL data and bimodal text-to-code data across six PL such as python, java and PHP. Additionally, data are collected from open-source repositories on GitHub. The inclusion of bimodal data significantly enables the model to enhance its understanding of PL together with NL.

2. Tokenizer: Tokenization is crucial for models like BERT and GPT. To handle Out-of-Vocabulary (OOV) tokens, most employ a Byte-Pair Encoding (BPE) tokenizer (Sennrich et al. 2016). CodeT5 adopts such a BPE tokenizer, setting its vocabulary size to 32,000, consistent with T5 (Radford et al. 2019). This tokenizer, trained on the model's pre-training data, efficiently filters out infrequent tokens. In comparison with T5's default tokenizer, CodeT5's variant reduces tokenized code sequence lengths by 30% - 45% to speed up training process. However, the default tokenizer in T5 struggled with some common symbols in code, such as misidentifying tokens like '{' and '}' as unknown (Wang et al. 2021).

With these modifications, CodeT5 becomes adept at code-related tasks. It undergoes further fine-tuning using the CodeXGLUE benchmark (Lu et al. 2021). The results indicate that CodeT5 significantly surpasses preceding methodologies, displaying superior performance in understanding tasks like code defect detection and clone detection. It also excels in generation tasks across domains such as code to code, code to text, and text to code, making it a prime choice for our experimental pursuits (Wang et al. 2021).

2.2 Evaluation Metrics in Text-to-Code

Text-to-Code is a subfield of NLP, focusing on tasks that involve converting NL to PL. The model should understand the NL input and generate corresponding PL as output. Evaluation metrics are used to measure the quality of the output.

2.2.1 BLEU score

Before the introduction of the BLEU score (Papineni et al. 2002), the evaluation of translation mainly relied on human judgment, which was both time-consuming and costly. This highlighted the increasing need for a machine-driven approach to evaluate translation outcomes.

In addressing this requirement, researchers followed the guiding principle that "The closer a machine translation is to a professional human translation, the better it is" (Papineni et al. 2002). At the core of the BLEU metric are two components: the modified n-gram precision and the brevity penalty (BP).

In the area of NLP, a "gram" means a single term in a sentence. N-grams, meanwhile, are continuous sequences of 'n' items extracted from a text sample. To illustrate, from the phrase "apple is a fruit", "apple" and "fruit" are 1-gram representations, while "apple is" and "is a" are examples of 2-grams. The use of n-grams serves to fragment the sentence into smaller, interconnected segments, offering a structural basis for comparisons with reference outputs. Though n-grams are widely used in machine translation, they come with inherent complexities (Papineni et al. 2002).

N-gram precision is generally computed by comparing the count of n-grams in a candidate translation with the maximum count of matching n-grams across all reference translations. However, this method has a potential drawback: if a candidate translation overuses certain terms, the resultant score can be misleadingly high. To solve this, the n-gram count from the candidate is "clipped" to the maximum observed count from the reference translations. After this modification, the sum of these clipped counts is divided by the total count of n-grams in the candidate translation to yield the modified n-gram precision, symbolized as P.

But a challenge emerges when relying solely on the modified n-gram precision: shorter sentences tend to receive disproportionately higher precision scores compared to longer ones. The brevity penalty (BP) was introduced to tackle this, and it's expressed as (Papineni et al. 2002):

$$\mathrm{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{2.6}$$

Where c denotes the length of the candidate translation, and r stands for the effective reference corpus length.

Combining the components that have been mentioned, the BLEU score is formulated as:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log P_n\right) \tag{2.7}$$

Where w_n are the weights, often set to 0.25 for n-grams ranging from 1-gram to 4-gram (Papineni et al. 2002). Notably, the 1-gram verifies content relevance by inspecting word overlaps, while longer n-grams gauge the coherence or fluency of the translation.

While the BLEU metric can't fully act as human assessment in terms of understanding and fluency judgment, its efficiency and cost-effectiveness make it an invaluable tool, especially in domains like machine translation.

2.2.2 CodeBLEU score

The BLEU metric, is typically used in machine translation tasks to evaluate the quality of text-to-text translations. However, when we transition from text translation to text-to-code generation tasks, the evaluation criteria must be modified. This is primarily because of the fundamental differences between NL and PL.

CodeBLEU emerges as a refined evaluation metric, built on the distinctive attributes of PL in mind (Ren et al. 2020). It relies on four essential components for its calculation: n-gram matching, weighted n-gram matching, syntactic AST (Abstract Syntax Tree) matching, and semantic data-flow matching. Traditionally, each of these components holds an equal weight of 0.25. The standard n-gram match gauges the accuracy and fluency of translations, a principle directly derived from BLEU (Papineni et al. 2002). But the distinctions between NL and PL, such as the variation in their vocabulary sizes, call for a different method. Due to PL have much less vocabulary sizes compared to NL, the weighted n-gram feature in CodeBLEU addresses this by giving more importance to keywords in PL.

In everyday conversations or stories, language typically follows a sequential pattern (Guo et al. 2019). On the other hand, PL are crafted for computer interpretation, resulting in a distinct structural format. This tree-like structure in PL is aptly named the Abstract Syntax Tree (AST), as depicted in Figure 2.4. The structure shown in Figure 2.4 focus on presenting the syntax of the code. It demonstrates how Python code is broken down into the AST format, clearly illustrating the relationship between different elements of the code (Guo et al. 2022).

CodeBLEU highlights the structural syntax of code sections using the AST match score:

$$Match_{ast} = \frac{Count_{clip}(T_{cand})}{Count(T_{ref})} \tag{2.8}$$

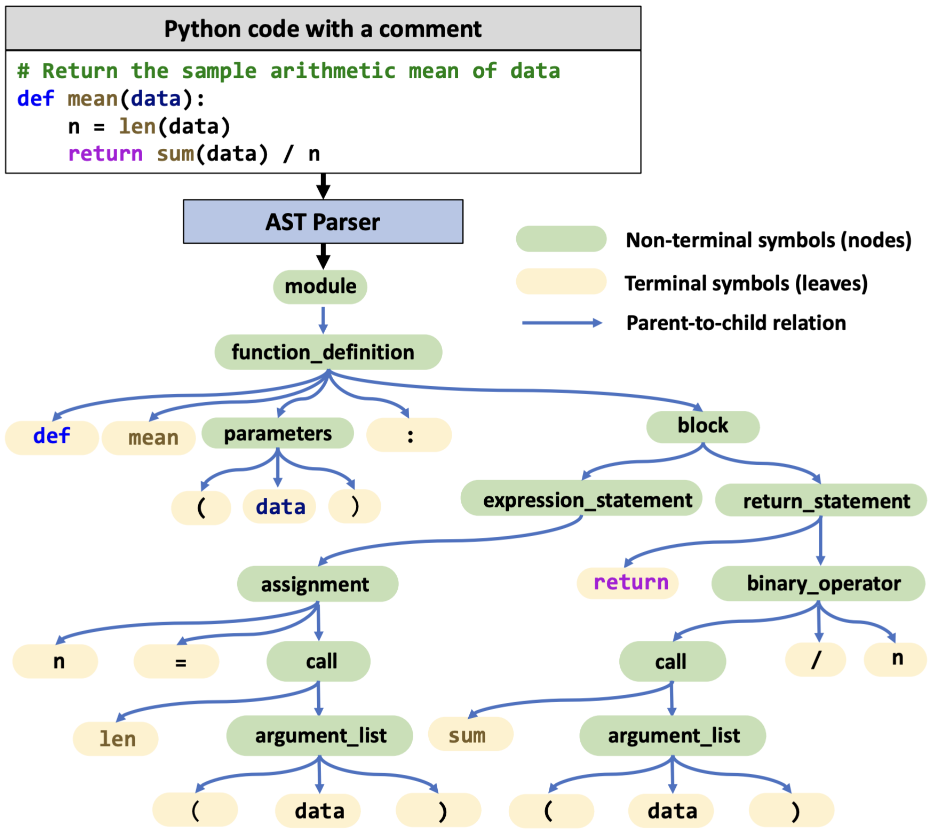
```
# Return the sample arithmetic mean of data
def mean(data):
    n = len(data)
    return sum(data) / n
```

Figure 2.4: AST representation for Python code (Guo et al. 2022)

Here, $Match_{ast}$ determines the extent to which the candidate AST segment, T_{cand}, aligns with the reference AST, T_{ref}. Difference between $Count_{clip}$ and $Count$ is that the former limits the count of n-grams in a candidate text to the maximum number of times it appears in the reference text. Essentially, this element evaluates the structural similarity between two pieces of code.

Data flow, represents the movement or transfer of data throughout different parts of a system or application. It's crucial to determine if the code executes correctly. Two functions may have resembling ASTs, but variations in their data flow can render them functionally different (Ren et al. 2020). CodeBLEU's semantic data-flow match score uses a method similar to AST matching but centers on the degree of matching data flows:

$$Match_{df} = \frac{Count_{clip}(DF_{cand})}{Count(DF_{ref})} \tag{2.9}$$

In this context, DF_{cand} and DF_{ref} represent the candidate and reference data flows, respectively. With this, CodeBLEU evaluates how functionally similar two code segments are.

In summary, CodeBLEU is developed considering the intrinsic properties of PL, making it more adapt for evaluation tasks like code-to-code or text-to-code generations.

Chapter 3

Related Work

In this chapter, we offer a comprehensive review of the text-to-code task, driving into both foundational concepts and recent developments in the field. Our overview includes a discussion on the underlying principles of pre-trained language models, with a deeper dive into models explicitly crafted for code generation tasks. Subsequent sections present data augmentation techniques set within the framework of NLP. Finally, we conclude with a discussion on related work concerning data augmentation in the code tasks domain.

3.1 Text-to-Code

In the introduction, we highlighted the importance of text-to-code generation. However, PL have far more precise requirements than NL in practice. Even if a model generates NL statements with grammatical errors, people can still understand the translation if it contains key information. In contrast, PL require computers to execute specific tasks; a single mistake can cause bugs.

The performance of the text-to-code task improves with the development of pre-trained language models like T5 (Raffel et al. 2020), BERT (Devlin et al. 2018), and GPT (Brown et al. 2020b). These models, built on the transformer architecture, have significantly improved performance benchmarks across a variety of NLP tasks. However, while these models showing good performance in diverse applications, they were not specifically developed for text-to-code generation. To address this gap, researchers introduced specialized models such as CodeT5 (Wang et al. 2021) and CodeBERT (Feng et al. 2020), refining foundational language models with code datasets. It's important to mention that most of these models undergo pre-training on vast unlabeled coding data and require fine-tuning for specific tasks.

Moreover, advanced language models tailored for code generation, such as StarCoder (Li et al. 2023) and Codex (Chen et al. 2021a), have been incorporated into software platforms like CodePilot. To support this effort, benchmark datasets including CodeXGLUE (Lu et al. 2021) and XLCOST (Zhu et al. 2022) have been introduced to the research community to evaluate model performance. Over time, additional datasets like HumanEval (Chen et al. 2021b), HumanEval-X which hand-craft improved from HumanEval (Zheng et al. 2023), MBPP (Austin et al. 2021), and MPTB have been developed specifically for testing text-to-code generation.

3.2 Data Augmentation

3.2.1 Overview

Advancements in model architecture have significantly improved in recent years. However, one persistent challenge has been the insufficiency of data for training these advanced models. This data shortage was first evident in the field of computer vision (CV). To address this, it's essential not only to find solutions for the data shortage but also to ensure that the models robust to noise that might exist when collecting data for training.

A solution to enhance model performance, especially when confronted with limited data, is data augmentation. Essentially, data augmentation refers to the process of generating new training samples by manipulating the existing data in the training dataset, without the need to collect additional data. In the field of CV, researchers initially adopted techniques such as flipping, rotation, translation, and noise injection to augment image data (Shorten and Khoshgoftaar 2019).

However, with the rapid evolution of NLP in recent years, the need for data augmentation methods specifically for text has increased. This need arises because language inherently presents distinct challenges when compared to images. While images exist in a continuous input space, NL operates in a discrete input space (Feng et al. 2021). Consequently, even a minor alteration, such as changing a single word in a sentence, can entirely shift its intended meaning.

Classical text data augmentation techniques can be categorized into two primary methods: adding noise to sentences and utilizing paraphrasing, such as back-translation. The noise-adding approach might involve methods such as substituting similar words or swap word positions within a sentence. On the other hand, back-translation involves translating a sentence into a pivot language and then translating it back to the original language, introducing changes in the process (Liu et al. 2020).

Building upon the idea of adding noise, it has been demonstrated that training models with noisy examples is conceptually analogous to employing Tikhonov regularization, which includes L2 regularization methods (Bishop 1995). Regularization can prevent model overfitting on the training data by introducing a penalty term. This can be instrumental in enhancing the model's performance on the test data. Moreover, from a theoretical view, data augmentation techniques can be linked to feature averaging processes and can also serve as tools for variance regularization (Dao et al. 2019). The feature averaging process ensures that the model doesn't over-rely on specific features in the training data, while variance regularization makes the model less sensitive to minor changes.

3.2.2 WordNet

Human dictionaries focus on providing word pronunciations, definitions, and usage examples. However, such an approach isn't readily adaptable for computerized language processing. WordNet (Miller 1995), in contrast, explains word meanings by linking words with related terms, particularly through synonyms. This method not only establishes relationships between words but also constructs a network, where each word is connected to others.

In WordNet, words are represented using ASCII characters. To offer a statistical perspective, this lexical database boasts over 118,000 unique word forms and more than 90,000 distinct word senses, aggregating to over 166,000 pairs. This extensive dataset makes WordNet an itool for tasks such as synonym replacement in data augmentation processes.

3.2.3 Easy Data Augmentation

Easy Data Augmentation (EDA) (Wei and Zou 2019) represents a set of data augmentation techniques designed primarily to introduce controlled noise into datasets. This intentional addition of noise serves as a countermeasure against model overfitting. Though the methods encapsulated within EDA are simple, they yield powerful results when applied. The four techniques encompassed by EDA include:

1. Synonym Replacement (SR): Choose n words randomly from the sentence and replace them with their synonyms. The words selected for replacement shouldn't be stop words. As mentioned in the previous section, synonym replacement typically leverages WordNet. This is because WordNet catalogues millions of synonym pairs, making it apt for data augmentation. Synonym replacement not only introduces noise but also transfers information from WordNet for data augmentation purposes.

2. Random Insertion (RI): Randomly select n words from the sentence and find their synonyms. Then, randomly insert these synonyms into the sentence. This method

complements the random deletion strategy to ensure that the length of the sentence, after applying the LSA method, remains in a range similar to the original.

3. Random Swap (RS): Randomly choose two words in the sentence and swap their positions. Repeat this n times. This method is typical for introducing noise into the dataset.

4. Random Deletion (RD): With a given probability p, randomly remove words from the sentence. This approach introduces noise into the dataset.

Operation	**Sentence**
None	Apple is a fruit that contains vitamin C.
SR	Apple is a fruit that incorporate vitamin C.
RI	Apple is a fruit that absolutely contains vitamin C.
RS	Apple is a fruit contains that vitamin C.
RD	Apple is a fruit that vitamin C.

Table 3.1: Sample sentence operations for data augmentation.

Though these methods may seem straightforward, their efficacy in text classification tasks is commendable. Wei and Zou (2019) showcased the results of data augmentation in LSTM-RNN and CNN. On average, across five datasets, training with EDA using only 50% of the available training set achieved comparable accuracy to using the entire set. The study also examined the performance decay for each EDA method with respect to the percentage of words replaced, denoted as α. All methods exhibited an initial improvement, peaking at $\alpha = 0.10$, after which performance typically began to decline. Notably, RD showed the most rapid decline since increasing α resulted in the loss of more information.

3.2.4 TF-IDF word replacement

Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical method used to measure the importance of a specific word in a document. It operates on the idea that if a word appears frequently in a particular sentence but is rare in the broader corpus, it might hold significant discriminatory power for tasks like classification.

1. **Term Frequency (TF)**: It quantifies how often a word appears in a given sentence relative to the most frequent word in that sentence (Rajaraman and Ullman 2011). The formula for TF is:

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}} \tag{3.1}$$

Where:

- f_{ij} is the frequency of term i in document j.
- $\max_k f_{kj}$ is the frequency of the most frequent term in document j. where k means any term from the set of all terms within document j.

2. **Inverse Document Frequency (IDF)**: IDF provides a measure of how much information a word provides by quantifying how frequently it appears across all documents (Rajaraman and Ullman 2011). The formula for IDF is:

$$IDF_i = \log_2 \left(\frac{N}{n_i} \right) \tag{3.2}$$

Where:

- N is the total number of documents in the corpus.
- n_i is the number of documents in which term i appears.

The product of TF and IDF gives the TF-IDF score of a word in a document:

$$TF - IDF_{ij} = TF_{ij} \times IDF_i \tag{3.3}$$

Understanding the principles behind this statistical method, the next step would be to use it for data augmentation. The goal is to introduce noise into the augmented dataset by replacing non-significant words, ensuring that essential information is not lost. Practically speaking, to ensure that the probability of replacing a word is negatively correlated with its TF-IDF score (Xie et al. 2020), we set the replacement probability as:

$$\min\left(p(C - TF - IDF_{ij})/Z, 1\right) \tag{3.4}$$

Where C is maximum TF-IDF score in a sentence and p is a hyperparameter controlling the extent of augmentation, and

$$Z = \frac{\sum(C - TF - IDF_{ij}))}{|f_j|} \tag{3.5}$$

represents the average score.

BERT tokenizes sentences into sequences of words and further divides these words into subwords. This approach is utilized to ensure that the generated examples are diverse and valid (Xie et al. 2020). The augmentation strategy is carefully designed to preserve key terms in a sentence while replacing less informative words with other non-informative alternatives.

When changing words in a sentence, we're careful to pick replacements that don't change the main idea. We look at all possible words but avoid key ones that are too important. Each word gets a new score based on how often it's used. This score helps us decide how likely a word is to be picked as a replacement.

In conclusion, The TF-IDF method identifies the importance of words in texts, guiding data augmentation processes. By utilizing BERT's tokenization, it ensures the augmented data remains diverse and meaningful. Strategic word replacements made via this approach enhance datasets by only replacing less important words in sentence.

3.2.5 Back-translation

Back-translation is a method of data augmentation that selects a pivot language to translate to, and then translates back to the original language (Sennrich et al. 2016). For instance, to enhance a sentence, one might use a pre-trained model to translate English to German, and then translate from German back to English.

The variety in linguistic expression shows the same idea can be articulated in various ways. Through the translation process, because the model used for translation and the one used for translating back can be different, we might obtain a sentence that's structurally different from the original but contains a similar meaning. The structural difference can be attributed to the distinct ways languages structure sentences. During augmentation, we should ensure that the meaning does not change, given that the corresponding label remains the same. This diversity is invaluable for training machine learning models as it enriches the data set.

The Back-Translation Procedure for original language tanslation is shown in Figure 3.1. The process involves translating text from original to a pivot Language using a pre-trained model, and then translating it back to original language type with another pre-trained model.

The pre-trained model employed in back-translation is a sequence-to-sequence model. Sennrich et al. (2016) developed neural translation systems for four language pairs for the WMT 2016 news translation task, leveraging an attentional encoder-decoder and BPE subword segmentation. They experimented with back-translations, pervasive dropout, and target-bidirectional models, achieving improvements of up to 11.2 BLEU over their base models. Nevertheless, as the architecture of NLP models has evolved, the accuracy of back-translation has enhanced owing to advancements in model architecture. Yang et al. (2020) in 2020 used transformers for data augmentation.

Although the quality of translations has significantly improved with the progression of

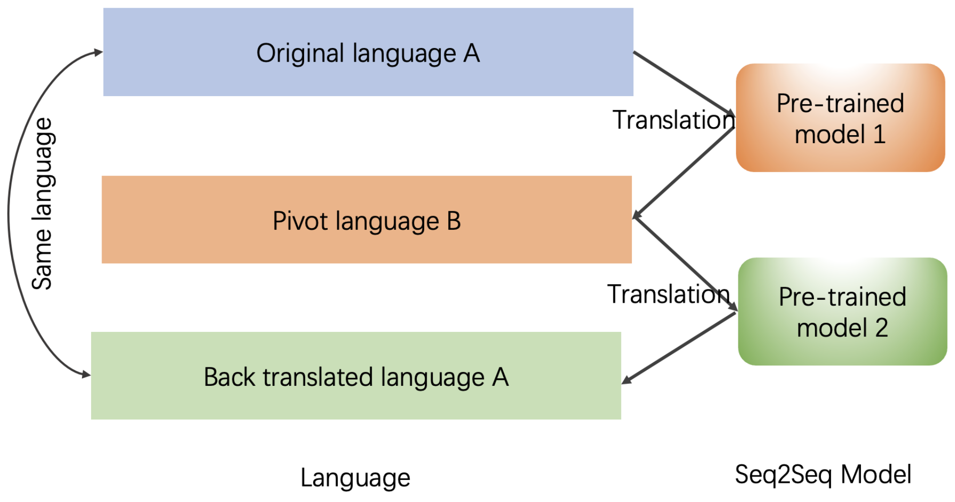

Figure 3.1: Back-Translation Procedure

neural network translation models (Liu et al. 2020), bringing the back-translated sentences closer in quality to the original texts, translations aren't always flawless. Back-translations can produce grammatically incorrect or semantically ambiguous sentences.

In conclusion, back-translation is a widely employed data augmentation technique in NLP. It capitalizes on the diversity of language, generating new sentence variants through the translation process, thereby offering more varied data for model training.

3.3 Data Augmentation in Code tasks

The exploration of data augmentation in the realm of code generation mainly involves tasks like code translation and text-to-code conversion. The augmentation techniques used in text-to-code tasks have mostly focused on modifying the code section (Chen and Lampouras 2023). It is relatively uncommon to see augmentation efforts aimed at the NL parts.

In this context, Yu et al. (2022) introduced a novel set of 18 data augmentation strategies, designed for operations like swapping and deletion, aiming to add noise to the code text. Their approach, as described in their work, is based on expanding large code datasets through a set of source-to-source transformation rules. These rules are carefully designed to not only maintain the code's semantics but also ensure its syntactic naturalness.

Ahmad et al. (2022) have highlighted the crucial role of back-translation in data augmentation and have applied it in the field of code generation. Their experimental procedure involves translating code into text and then converting that text back into its original code form, aiding data augmentation for code translation tasks. From the tests, it seems that even if the pre-trained models are not performing well at back-translation at first, we can train them in a special way, using code summarization and generation to perform back-translation, to help them work with multiple languages. With more training in back-translation, they get even better at converting one PL to another.

In the quest to enhance data augmentation for code translation, one can skip the step of translating into NL. For example, to augment data for translating from PL_A to PL_B, one might first train a model to go from PL_B to PL_A. Using this model, monolingual data in PL_B can be converted into a version, let's call it PL_{A0}, forming pseudo-parallel pairs $PL_{A0} - PL_B$. Such pairs then help in further model training. This method, explored by Chen and Lampouras (2023), offers potential advantages, especially considering the possible noise and errors from two-way translations.

Another beneficial area to exploit data augmentation is in the task of code summarization. Instead of turning to external datasets or models for back-translation, a fresh approach is to "reverse" the existing dataset, using it in a multilingual setting that includes different PL (Chen and Lampouras 2023). When tested by CodeXGLUE (Lu et al. 2021), results show that CodeBERT (Feng et al. 2020) demonstrates noticeable improvements in both tasks, with increases in code translation for 6.9% and summarization tasks for 7.5%.

Chapter 4

Methodology

In this chapter, we categorize various augmentation methods into three primary categories: word replacement, noise addition, and paraphrasing. Initially, we will describe these methods, followed by the application of diverse data augmentation techniques and the fine-tuning of the model to identify the most effective data augmentation method for the text-to-code task, using the XLCoST dataset as a benchmark. Subsequently, we will delve deeper into the analysis of each data augmentation method by selecting a sample and comparing it with the original text, thereby enabling a more comprehensive evaluation of its performance. Finally, we will draw conclusions and highlight the limitations of our study.

4.1 Dataset Overview: XLCoST

XLCoST is a benchmark dataset tailored for different programing languages. The XLCoST dataset is divided into two main parts: the Snippet-level and the Program-level. The first part is the Snippet-level. With data sizes of 446K/22K/41K corresponding to the train/dev/test set, respectively, it focuses on generating a code snippet from a given comment. This is the size of the total dataset across all languages. The Snippet-level dataset establishes a new standard for cross-lingual code intelligence, making it effective for both code translation and code generation tasks. This dataset offers detailed parallel data across 8 programming languages: C++, Java, Python (Py), C#, JavaScript (JS), PHP, and C.

The second part is the Program-level. It comprises data sizes of 50K/3K/5K for all languages. This task requires to generate a complete program based on a provided problem description and its associated comments.

In this thesis, we are mostly interested in the program-level for the following reason: Data associated with program-level is longer in NL than that of snippet-level, which shown in Table 4.1. This difference in length results from the combination of problem descriptions

Type	**Content**
Text ('python-snippet')	Python3 implementation of the above approach
Code ('python-snippet')	`def maxPresum ( a , b ) :  NEW_LINE`
Text ('python-program')	Check if a number can be represented as sum of two positive perfect cubes — Python3 program for the above approach ; Function to check if N can be represented as sum of two perfect cubes or not ; If it is same return true ; ; If the curr smaller than n increment the lo ; If the curr is greater than curr decrement the hi ; Driver Code ; Function call to check if N can be represented as sum of two perfect cubes or not
Code ('python-program')	`import math NEW_LINE def sumOfTwoCubes (n) : NEW_LINE INDENT lo = 1 NEW_LINE hi = round (math . pow (n , 1 / 3)) NEW_LINE while (lo <= hi) : NEW_LINE INDENT curr = (lo * lo * lo + hi * hi * hi) NEW_LINE if (curr == n) : NEW_LINE INDENT return True NEW_LINE DEDENT if (curr < n) : NEW_LINE INDENT lo += 1 NEW_LINE DEDENT else : NEW_LINE INDENT hi -= 1 NEW_LINE DEDENT DEDENT return False NEW_LINE DEDENT N = 28 NEW_LINE if (sumOfTwoCubes (N)) : NEW_LINE INDENT print (" True ") NEW_LINE DEDENT else : NEW_LINE INDENT print (" False ") NEW_LINE DEDENT`

Table 4.1: Samples from the XLCoST dataset.

and code comments. Extended comments in Program-level dataset offers more context for data augmentation, giving us greater flexibility for methods like word substitution and noise addition, thus making the model more reliable.

Our preference for this dataset is based on two main reasons. Firstly, the clarity of the NL content is essential. For efficient data augmentation, it is imperative to ensure the NL data is easily understood. Contrarily, datasets from sources such as CodexGLUE's CONCODE can be difficult to comprehend (Lu et al. 2021).

From the above comparison, it is evident that the XLCoST dataset offers a more comprehensible representation, making it a preferable choice for data augmentation analysis. The Python programming language was selected as our main focus, given my proficiency in Python programming.

We selected the "python-program" dataset[1] which is available on HuggingFace. This subset consists of 9,263 training samples, 472 validation samples, and 887 test samples. We

[1] https://huggingface.co/datasets/codeparrot/xlcost-text-to-code

chose this subset because it has already been preprocessed, making it straightforward for downloading and further modification.

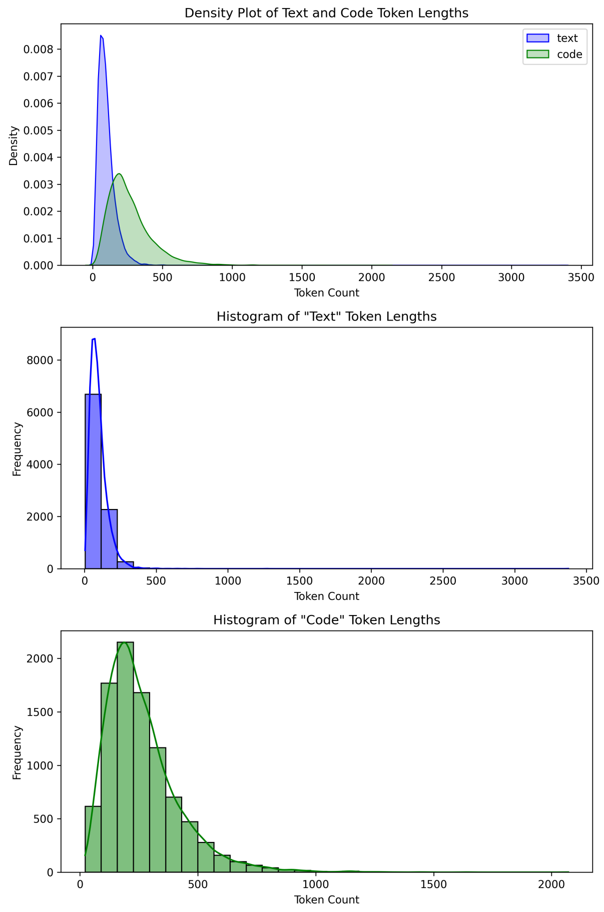

Figure 4.1: Statistics of the dataset for python-program in XLCoST.

The overall statistics for this dataset are shown in Figure 4.1. Based on the provided data, the average token length for 'text' is approximately 96.89, with a median of 84 tokens. In contrast, 'code' has an average token length of approximately 264.82 and a median of 230 tokens. The spread of token lengths for 'code' is considerably longer than that of 'text'. Specifically, most 'text' entries have fewer than 200 tokens, while 'code' entries typically fall below 400 tokens.

Delving into an illustrative example, a specific dataset is presented in Table 4.1. This dataset uses special placeholders such as "NEWLINE", "INDENT", and "DEDENT". These placeholders are representative of structural formatting in code: "NEWLINE" represents a new line, "INDENT" suggests an increase in indentation level by 4 spaces, and "DEDENT" indicates a decrease in indentation level by 4 spaces.

4.2 Model Choice

Existing models have been evaluated on this dataset by prior work (Zhu et al. 2022), and it appears that the CodeT5 model is performing best across the board (Wang et al. 2021). In more detail, CodeBERT (Feng et al. 2020) consistently underperformed compared to PLBART (Ahmad et al. 2021) and CodeT5 across the majority of generation tasks, as indicated in the initial sections of the table. Notably, both PLBART and CodeT5 (Wang et al. 2021) build by encoder-decoder architectures that are pre-trained using sequence-to-sequence objectives. In contrast, only the encoder in CodeBERT has undergone pre-training, leaving the decoder weights to be randomly initialized for such tasks (Zhu et al. 2022). This distinction in architectures might explain the observable difference in performance across the models.

The overall architecture and pre-training techniques are extensively detailed in Chapter 3. For our study, we employed the CodeT5-base model [2], which has 220 million parameters. This model is not only compatible with our computational constraints but also demonstrated superior performance in the dataset's research paper (Zhu et al. 2022).

According to the paper's results on the Python program dataset which we used in this thesis, the CodeT5-base model was fine-tuned over 10 epochs and achieved a score of 35.02. This surpasses the scores of PLBART and CodeBERT, which registered 33.77 and 24.5 respectively. It's notable that the remaining hyperparameters were retained as in the original pre-trained model, ensuring consistency in comparison (Zhu et al. 2022).

4.3 Fine-tuning Strategy

For our fine-tuning strategy, we utilized the training data described in the previous section, titled "Python-program-level". Further comparison was made to fine-tune the combination of training data and the amount of augmented datasets. Based on the previous dataset analysis for NL, the input and target lengths were each limited to 400 tokens. Each token approximately corresponds to 1.3 English words.

The training configuration was carefully designed, employing a learning rate of 5×10^{-5}. During training, each batch consisted of 16 samples, and this uniformity was maintained for both training and evaluation phases. Regularization techniques, such as weight decay set at 0.01, were applied. All these settings are consistent with Zhu et al. (2022). However, enhanced computational efficiency was achieved using gradient checkpointing and FP16 precision.

[2]https://huggingface.co/Salesforce/codet5-base

The optimizer we utilized is AdamW (Loshchilov and Hutter 2017), which is an improved version of the Adam optimizer (Kingma and Ba 2014). AdamW introduces a modification to the weight decay regularization, separating the weight decay from the optimization steps. This allows for a more appropriate weight decay process that does not interfere with the adaptive learning rate computation, hence lead to improvement in ability to handle sparse gradients more effectively.

Another distinguishing factor is the training spanned a maximum of 15 epochs. We saved the model every 1000 steps, which it was also evaluated using the cross-entropy loss on the validation. The solution we proposed involves selecting the best model based on the lowest cross-entropy loss for further evaluation.

Within our training framework, early stopping was integrated as a preventive measure against overfitting and to conserve computational resources. This mechanism would break the training process if the validation loss failed to show improvements over three consecutive evaluations. With these configurations in place, we employed the 'Trainer' class from the Hugging Face Transformers library to initiate and manage the training process (Wolf et al. 2020).

4.4 Data augmentation method

After introducing the baseline and the fine-tuning process, the subsequent step involves employing data augmentation techniques to enhance the performance of the fine-tuned model relative to the baseline. As previously discussed in the literature review, there are various methods available to explore, and their results can guide our selection. The augmentation techniques we employed are grouped into three categories: word replacement, noise addition, and paraphrasing.

4.4.1 Word replacement

Starting by introducing word replacement, the meaning of word replacement involves selecting certain words in a sentence and replacing them with other words. There are four augmentation methods that can be classified under this group: using WordNet for synonym replacement, using Word2Vec for synonym replacement, using a pre-trained BERT model to predict masked words, and using TF-IDF to replace words that are not important in the sentence.

WordNet

The first method under consideration is WordNet-based synonym replacement (Miller 1995). This technique based on WordNet, a substantial lexical database of English, to serve as a dictionary for computers. When a user consults this dictionary, synonyms for the target word can be identified, facilitating its replacement. A significant advantage of this method is its efficiency; compared to other techniques, it does not rely on any pre-trained model, thereby offering cost-free inferences. This method derives from the Easy Data Augmentation (EDA) strategy (Wei and Zou 2019). Previous literature has already highlighted its potential in enhancing model performance for classification tasks.

Word2Vec

Following is the Word2Vec-based synonym replacement method. Relying on the Word2Vec model (Mikolov et al. 2013), which understands the meanings of words from how the models predicts words based on their surrounding context, this strategy substitutes a word with another associated a similar embedding vector. For the purposes of this thesis, the specific model employed for Word2Vec is "word2vec-google-news-300"—a model that harnesses the skip-gram algorithm and is pre-trained on a part of the Google News dataset comprising roughly 100 billion words (Mikolov et al. 2013)[3]. The "300" denotes its 300-dimensional nature. Opting for a skip-gram trained model is beneficial for word replacement, since it excels at capturing the dynamics between a word and its surrounding context. Consequently, words that share analogous contexts tend to have closely aligned embeddings within the vector space, making this method likely superior to the simpler synonym replacement facilitated by WordNet.

BERT prediction

This technique is based on a pre-trained BERT-Base model (Devlin et al. 2018). A fundamental characteristic of the BERT model is its training using masked words, with approximately 15% of words being masked during pre-training. Given this behavior, BERT is adept at predicting the masked word based on the surrounding context. For data augmentation, this property can be harnessed by randomly masking certain words in a sentence, feeding the altered sentence into BERT, and having the model predict the masked word. The ensuing prediction can be used as a replacement, ensuring that the broader context around the masked word is taken into account during the substitution.

[3] https://huggingface.co/fse/word2vec-google-news-300

TF-IDF

In this method, the Term Frequency-Inverse Document Frequency (TF-IDF) metric is employed to ascertain the relative importance of words within a sentence. Initially, a TF-IDF model must be trained on the entire dataset. Subsequently, individual sentences are fitted to identify words with lower TF-IDF scores. Words with reduced scores are deemed less crucial in the context of the broader document or dataset. Thus, these less significant words are targeted for replacement by introducing random character errors. This approach introduces noise into the dataset, but the inherent meaning of the NL data remains mostly unchanged.

4.4.2 Noise addition

In the realm of data augmentation, adding noise has been extensively employed in NLP. One notable approach is the EDA method, which encompasses random word replacement, insertion, and deletion. This method has demonstrated its effectiveness, particularly in sentiment classification tasks (Wei and Zou 2019). Another work by Yu et al. (2022) has effectively employed word swapping and deletion techniques for code-to-code generation tasks. Inspired by these methodologies, we aim to employ two distinct strategies. For each technique, we plan to adjust the magnitude of augmentation and its possibility of occurrence to optimize the results for data augmentation.

Word Swap and Deletion

The word swapping proposed in this research differs from the random two words swapping technique of EDA (Wei and Zou 2019). Instead, we focus on swapping adjacent words within a sentence. Such a method has been used by other research as well (Belinkov and Bisk 2017; Yu et al. 2022). We believe that random swapping of two words in a sentence can potentially distort its meaning. However, this risk might decrease when the swap is limited to adjacent words.

Deletion, on the other hand, involves the random removal of words. This technique has been recognized as an effective data augmentation strategy (Pruthi et al. 2019). However, its efficacy is dependent on the deletion probability. While a lower probability can enhance a model's performance, increasing this probability can adversely impact the results.

Character Swap

Rather than focusing solely on words, we can delve deeper into the linguistic structure by considering characters for augmentation. Augmentation at the character level can be categorized into four main techniques:

1. Swap: Interchanging two adjacent characters within a word.
2. Substitution: Replacing a character in a word with another random character.
3. Deletion: Removing a random character from a word.
4. Insertion: Introducing a random character within a word.

We believe that character-level augmentation is a way of introducing noise without causing significant misinterpretations by the model. Also, this method mimics real-world cases where people often do typing errors.

4.4.3 Paraphrasing

This section delves into the area of paraphrasing and highlights two methods. The first method, known as back-translation, has gained significant popularity and is commonly used for sequence-to-sequence data augmentation, especially in code-related tasks (Ahmad et al. 2022; Gupta et al. 2021; Chen and Lampouras 2023). The second approach involves utilizing the GPT-3 model to paraphrase sentences, aiming to produce semantically similar outcomes.

Back-translation

The back-translation technique employs pre-trained models sourced from "Helsinki-NLP/opus-mt"[4]. These types of models were selected because they were pre-trained on several versions that support a variety of languages and have been translated in both directions. Hence, we selected German, French, and Spanish as our pivot languages. The reason behind this selection is that all three languages have roots in the Latin language family, ensuring that the essence of the sentences remains largely unchanged even after translation. To implement this method, six pre-trained models were used. For each language, one model translates to the pivot language and another translates it back to the original language. The efficacy of these pre-trained models is gauged using the BLEU score metric (Papineni et al. 2002), with their performances with given test dataset tabulated in Table 4.2.

From the table, analyzing the performance of the pre-trained models is difficult, primarily because each model has been evaluated on distinct text datasets. Nevertheless, the reliability of these models can be affirmed since they rank among the most downloads models on HuggingFace for language translation.

[4] https://huggingface.co/Helsinki-NLP/opus-mt-en-es

pre-trained Model	**Test Dataset**	**BLEU**
English to German	Tatoeba.en.de	47.3
German to English	Tatoeba.de.en	55.4
English to French	Tatoeba.en.fr	50.5
French to English	Tatoeba.fr.en	57.5
English to Spanish	Tatoeba-test.eng.spa	54.9
Spanish to English	Tatoeba-test.spa.eng	59.6

Table 4.2: BLEU Scores of pre-trained Models used for Back-translation ().

GPT Paraphrasing

GPT-3 stands out as one of the dominant large language models (LLM) in recent times (Brown et al. 2020b). With an impressive arsenal of roughly 175 billion parameters, it exhibits exemplary proficiency, especially in English-centric tasks. This potency led us to consider its potential for English paraphrasing. The specific variant chosen for our study is "gpt-3.5-turbo"[5]. However, there were challenges when trying to employ this model. The nature of our input, which is typically instructive for code generation, occasionally results in the model overlooking the paraphrasing intent and directly producing Python code. To overcome this, specific prompts were used, as shown below in Table 4.3.

Role	**Content**
System	You are a sophisticated English translator tasked with paraphrasing the following text. Remember, your goal is to provide a different way of saying the same thing, not to generate code or other responses.
User	Paraphrase + *Natural Language Problem Description*

Table 4.3: Specific prompts used for "gpt-3.5-turbo" to guide paraphrasing without code generation.

In this method, we specify the role of GPT as a specialized English translator. A significant emphasis was placed on the prompt "paraphrase" within the user's content to ensure the model refrained from direct code generation. However, this introduced an unintended consequence: the model exhibited a propensity to rephrase the word "paraphrase" at the outset of sentences. Consequently, we instituted a data cleaning process, removing this term and other semantically similar terms to "paraphrasing", ensuring cleaner output.

[5]Procedure for accessing this model through its API can be found at https://platform.openai.com/docs/models/gpt-3-5

4.5 Experimental results

After the data was augmented using the previously mentioned methods, we began the fine-tuning procedure. For each iteration, the “codeT5-base” model was fine-tuned using a combined dataset: its full original training set and the augmented dataset. Our approach to the augmentation process was systematic.

We experimented with varying probabilities of replacement and noise, starting at 15% and extending to 30%. The initial choice of 15% was influenced by models like BERT, which were trained as Masked Language Models using a 15% masking rate; we anticipated that our augmented dataset might perform optimally at this percentage. Our range was determined to strike the right balance between preserving the original intent of the data and introducing meaningful variations through augmentation. If our preliminary results indicated a potential for improved performance, we conducted further iterations with adjusted probabilities.

Regarding the augmentation amount, the majority of our experiments were carried out with factors of 1 and 3. This means the size of the augmented dataset was either the same as or three times the size of the original dataset. However, for methods like paraphrasing, the strategy differed due to the intrinsic nature of the method. In such cases, we maintained a consistent augmentation amount across different languages.

The resultant performance metrics, as detailed in Table 4.4, give an overview of the effectiveness for each augmentation method combined with the fine-tuning process.

In the table, the BLEU score is mainly for reference. CodeBLEU is an evaluation metric designed specifically to measure the quality of code outputs. We first introduce the model, without any fine-tuning, meaning we test the pre-trained CodeT5 model as it is, it achieves a CodeBLEU score of 22.72%, considerably lower than the baseline score of 37.99%. In the remainder of this section, we will primarily compare the results with the baseline model.

For the Word Replacement methods, using WordNet (Miller 1995) with a replacement probability of 15% slightly surpasses the baseline with a CodeBLEU (Ren et al. 2020) score of 38.46%. However, increasing the probability to 30% results in a slightly worse performance, achieving a score of 38.02%. When employing the Word2Vec pre-trained model to replace synonyms, the results hover around the baseline, peaking at a CodeBLEU score of 38.48% with an augmentation amount of 3 augmented instances and a probability of 30%. The method using the pre-trained BERT base model (Devlin et al. 2018), despite varying probabilities and augmentation amounts, yields scores close to the baseline, with the highest being 38.32% at a probability of 15%. Interestingly, the TF-IDF approach is consistently below the baseline results, regardless of the replacement probability.

For the Noise Addition methods, the Word Swap and Deletion technique shows improve-

Model	**Aug. Amount**	**Probability**	**CodeBLEU (%)**	**Δ(%)**	**BLEU (%)**
Without fine-tune	-	-	22.72	-15.27	0.02
Baseline	-	-	37.99	-	32.03
Word Replacement					
Wordnet	1	0.15	38.46	0.47	32.09
	1	0.3	38.02	0.03	31.73
Word2Vec (Synonym)	1	0.15	37.79	-0.20	31.62
	1	0.3	38.25	0.26	31.92
	3	0.3	38.48	0.49	32.13
BERT(mask)	1	0.15	38.32	0.33	32.27
	1	0.3	38.20	0.21	31.77
	3	0.3	37.76	-0.23	31.31
TF-IDF	1	0.15	37.80	-0.19	31.31
	1	0.3	37.87	-0.12	31.54
Noise Addition					
Word Swap and Deletion	1	0.15	37.82	-0.17	31.61
	1	0.3	38.47	0.48	32.09
	3	0.3	38.77	0.78	32.74
	3	0.5	38.42	0.43	32.39
Char replacement	1	0.15	38.47	0.48	31.92
	1	0.3	38.37	0.38	31.84
	3	0.3	38.38	0.39	32.01
Paraphrasing					
GPT3	1	-	38.69	0.70	32.20
Back-translation					
German	1	-	**38.84**	**0.85**	**32.78**
French	1	-	38.51	0.52	32.39
Spanish	1	-	38.73	0.74	32.40

Table 4.4: Experimental results for various data augmentation methods.

ment. At a replacement probability of 15%, it achieves a CodeBLEU score of 37.82%. By increasing this probability to 30% and using an augmentation amount of 3, the score improves to 38.77%, notably surpassing the baseline and ranking as the second highest amongst all results. However, when we increase the probability to 50%, the results for this method decreased rapidly to 38.42%. The Char replacement method, on the other hand, consistently hovers around the baseline across various parameters, achieving its highest CodeBLEU score of 38.47% at a probability of 15% for a single augmentation.

Regarding the Paraphrasing methods, their performance is the best across all methods. Specifically, the Back-translation approach demonstrates significant results. Translation into German yields a CodeBLEU score of 38.84%, making it one of the highest-performing techniques relative to the baseline. Both French and Spanish back-translations also perform well, achieving CodeBLEU scores of 38.51% and 38.73% respectively. The method using GPT-3 for paraphrasing (Brown et al. 2020b), with a score of 38.69%, aligns closely with the results from other high-performing paraphrasing techniques.

In conclusion, while various augmentation techniques and their parameters produce a range of results, many consistently approach or are under the baseline CodeBLEU score. Techniques like paraphrasing stand out, whereas word replacement generally underperforms. However, the improvement in the best-performing result compared to the baseline is approximately 0.85%.

4.6 Discussion

For further analysis of these results, we need to delve deeper into each data augmentation method, using several examples for further explanation. In this section, each of the three categories: word replacement, noise addition, and paraphrasing, will be discussed separately at the beginning, with an overall conclusion provided at the end.

4.6.1 Word replacement

From Table 4.5, it is apparent that the replacement using the WordNet method seems nonsensible as it does not consider the overall meaning of the sentence. However, there is an instance where it replaces "be" with "represent", which, although semantically similar, is not an acceptable substitution because the subsequent word is "represented" as shown in 4.5. This highlights a limitation of the WordNet method; it only delves into the meanings of individual words in the dictionary without regarding the context of the word in the sentence. Despite this, the method can still yield readable augmented results, and the fine-tuned model results indicate it offers the best outcomes among all word replacement methods.

While the pre-trained Word2Vec replacement example appears to be quite noisy, the main issues with this augmentation can be grouped into three categories. First, there is the alteration of numerical values. For instance, the original text contains "two", but the Word2Vec model substitutes it with "three". This substitution is based on the closeness of these numerical words, leading the model to presume a strong association and thus make the replacement. Such an approach can lead to associating a code program with an incorrect natural language description.

Method	Sentence
Original NL	"Check if a number can be represented as sum of two positive perfect cubes — Python3 program for the above approach; Function to check if N can be represented as sum of two perfect cubes or not; If it is same return true; ; If the curr smaller than n increment the lo; If the curr is greater than curr decrement the hi; Driver Code; Function call to check if N can be represented as sum of two perfect cubes or not"
WordNet(p=0.15)	"Check if a number can represent represented as sum of two positive perfect cubes — Python3 program for the above approach; Function to check if N can be represented as sum of two perfect cubes or not; If it is same return true; ; If the curr smaller than n increment the lo; If the curr be greater than curr decrement the hi; Number one wood Code; Function squall to check if N can be represented as sum of two perfect cubes or not"
Word2Vec(p=0.3)	"Find unless a numbers could be represented as sum of three positive ideal cube — Python3 program forthe this below approaches ; Functions to check if N can be representing as sum of three perfect cubes or not ; If that was Same return true ; ; If the curr smaller more n increments the lo ; If this Automatic_Fuel_Injected_SILVE is larger more Automatic_Fuel_Injected_SILVE decrements the hi ; Driver Code ; Function call to check if N can be represented as sum of two perfect cubes or not"
BERT(p=0.3)	"Check if a number can been represented as sum of three positive perfect cubes — Python3 program for the above approach. Function to check whether N can be represented in sum between two perfect cubes or not ; If it the same as true hi ; if the curr and hi increment the lo ; If the hi is less than the decrement of hi ; Driver Code ; ; call to check whether n can be represented as sum of two Perfect cubes and not"
TF-IDF(p=0.3)	"Check if topleft can be represented as sum Stooge two positive perfect cubes Python3 program for the above radiations Function to check if can be represented as sum of two perfect cubes or not XX it is same return true If the curr smaller than Example pair1 lo If the curr is greater bomb curr decrement the hi Anagram Code Function call to check if can be represented confirms sum Economical two perfect cubes or not"

Table 4.5: Sentences after data augmentation using word replacement methods.

Second, there is the introduction of unusual noise, such as "Automatic_Fuel_Injected." This likely occurred because Word2Vec is an older model, and it utilized the Google News dataset for unsupervised learning. The NL used in news articles differs considerably from the NL used for code generation, potentially resulting in noisy output errors.

Lastly, some replacements are too tiny to be effective; for example, the model changes "same" to "Same," just changing the capitalization without addressing our goal of changing the synonym. This is also evident in the model's tendency to change words to their plural or past tense forms.

BERT exhibits problems similar to Word2Vec, such as replacing key figures and making minor modifications instead of altering synonyms. These issues may be common across all pre-trained models. Unlike Word2Vec, BERT does not generate noise, likely because it is a newer model with more diverse pre-trained source material. While BERT predicts words using a mask and often selects the word with the highest probability from its output list, other words from the list might be more contextually appropriate. As a result, the augmented results from BERT model is close to those from the Word2Vec model.

TF-IDF is a popular choice for classification tasks, but for sequence-to-sequence tasks, a more precise augmentation method is required due to the complexity of the output. This is especially true for code generation tasks, which demand exceptionally accurate training datasets. The replacements made by the TF-IDF method lack logical coherence and readability, resulting in fine-tuned results that are lower than baseline, regardless of the replacement probability.

4.6.2 Noise addition

Table 4.6 is showing the example for Noise adding method, at here, we are going to describe method for Word swap and delete, with character replace separately. From Table 4.6, it is clear that the word swap and delete method has significant limitations. It does not consider the overall sentence structure and semantics, leading to grammatical errors and changes in sentence meaning. For example, it swaps the position of "to" and "check", making the sentence grammatically incorrect, and changes "if the curr smaller than n increment the lo" to "if the curr smaller than increment n the lo," altering the sentence's meaning.

Similarly, the character replacement method replaces individual characters, resulting in mostly incorrect and nonsensical sentences, like replacing "Check" with "CTheck," and "number" with "unmber."

Despite these limitations, adding noise through word replacement has proven to be useful in code generation tasks. For instance, with a 30% augmentation probability, the method

Method	**Sentence**
Word swap and delete	"Check if a number can be represented as sum of two positive perfect cubes — Python3 program for the above approach; Function check to if N can be represented as sum of two perfect cubes or not; it is If same true return; ; If the curr smaller than increment n the lo; If curr the is than greater curr decrement the hi; Driver Code; Function call to check if N can represented be as sum of two perfect cubes or not"
Char replace(p=0.15)	"CTheck if a unmber can be represented as um of tgwo poitive perefct cubes — Python3 program for the above approach ; Function to check if N can be represented as osum of two perfect cubes or not ; If it is same return ture ; ; Ic the curr smaller than n incremnt the lo ; HIf the curr is greater than crur decrement the hi ; Driver Code ; Function call to check if N can be represented as sum of two perfect cubes or not"

Table 4.6: Sentences after data augmentation using Noise addition methods.

achieved a CodeBLEU score of 38.77, indicating its effectiveness up to a point (Yu et al. 2022). However, increasing the probability to 50% caused a rapid decline in performance due to excessive noise and the random deletion of key information. Similar tendency also shown in using EDA for classification tasks (Wei and Zou 2019).

On the contrary, character replacement, which completely changes word meanings, did not yield better results as swapping words did. Although this method might better mimic real world scenarios, where humans often mistype, the accuracy of our test set does not align with this reality, resulting in suboptimal model performance.

4.6.3 Paraphrasing

Table 4.7 delves into the effectiveness of back-translation as an approach for this particular task. Contrary to word replacement methods, back-translation substitutes suitable words without altering the meaning of the sentence, and crucially, does not modify important figures or Python commands. For instance, the back-translation method involving German translates the original sentence into German and then reverts it back to English, yielding a paraphrased sentence that preserves the original meaning and structure, as well as key Python commands like "return true" and "increment the lo."

This is further substantiated by the results of the fine-tuned model, where the model trained using German back-translation achieved the highest CodeBLEU score of 38.84%. A similar approach utilizing French as pivot language also yielded optimal results.

On the other hand, GPT-3 paraphrasing tends to alter the language structure more

Method	**Sentence**
Back-translation (German)	"Check if a number can be represented as a sum of two positive perfect cubes.Python3 program for the above approach; function to check if N can be represented as a sum of two perfect cubes or not; If it is the same, return true ; If the currency is smaller than n increase the lo ; If the currency is larger than curr Decrement the hi ; driver code ; function call to check if N can be represented as a sum of two perfect cubes or not"
Back-translation (French)	"Check whether a number can be represented by the sum of two perfect positive cubes of the Python3 program for the above approach; Check whether N can be represented by the sum of two perfect cubes or not; If it is the same true return; If the curr smaller than n increments the lo; If the curr is greater than the curr decrement the hi; Code of Conduct; Call function to check whether N can be represented by the sum of two perfect cubes or not"
GPT-3 paraphrasing	"Python3 program to determine if a number can be expressed as the sum of two positive perfect cubes. The function checks if N can be represented as the sum of two perfect cubes. If it is possible, the function returns true. The function uses the following approach: if the current number is smaller than N, increment the lower bound; if the current number is greater than N, decrement the upper bound. The driver code includes a function call to check if N can be represented as the sum of two perfect cubes or not."

Table 4.7: Sentences after data augmentation using paraphrasing.

significantly, as observed in the transformation of the original sentence to "Python3 program to determine if a number can be expressed as the sum of two positive perfect cubes..." While this modification maintains the original meaning, it significantly changes the sentence structure compared to back-translation. This may not always be desirable, especially for tasks that necessitate the preservation of specific language structures or technical commands. In situations where GPT-3 meets such scenarios, it sometimes results in the loss of key information. However, the results for GPT-3 paraphrasing are average in terms of back-translation and superior to most other methods, indicating its utility as an augmentation technique.

4.6.4 Overview and Limitations

The advantages and drawbacks of each augmentation method were presented with clear examples. We discovered that most of the data augmentation methods, except for TF-IDF, can yield positive results compared to the baseline. Those verified that the paraphrasing method overall performs best, and the method for swapping words and deleting could achieve an optimal solution when the amount of augmentation and probability are optimal. Other methods could still improve the model's performance within a certain range. However, the thesis objective is not only to compare each augmentation technique's effectiveness and make observations, but also to improve the code model's performance.

Nonetheless, the performance of the fine-tuned model only improved by 0.85% compared to the baseline with the best augmentation method, indicating that it probably is not a significant improvement even after several data augmentation methods were applied. After careful consideration, two underlying possible reasons were identified.

The first one is that data augmentation is actually challenging for tasks such as text-to-code generation. NL needs to be precise enough for generating code; any minor change could have a negative influence on the procedure of code generation. Adding noise to NL may not be as helpful as in tasks such as classification or machine translation due to the strict format and grammar requirements of code.

Another reason could be due to the data quality. The quality of a dataset can be considered as its informativeness and clarity. Although the CodeT5 model is well pre-trained for code tasks, the "python-program" dataset from XLCoST is considerably lengthy. Other experiments tend to use snippet-level data for code generation tasks (Chen and Lampouras 2023), where using snippet-level data could enable the model to generate shorter outputs and hence achieve better performance. If the model's performance is better, the improvement in the augmentation results can also be proportionally magnified. However, we aimed to use datasets with longer NL texts to enhance data augmentation visualization.

This indicates that different datasets might yield varying outcomes based on their levels of informativeness and cleanliness. Beyond inherent issues in the dataset itself, there is also the possibility that these NL augmentation methods may not be particularly effective for this specific dataset.

Chapter 5

Analyzing Back-translation

In this chapter, we start by visualizing the effectiveness of back-translation using only half of the dataset. We find that applying this augmentation technique to half of the dataset does not necessarily lead to positive experimental outcomes. Delving further into the full dataset, we present a learning curve that displays the CodeBLEU scores for various scales of back-translated augmented datasets. By using 75% of the back-translated dataset with German as the pivot language, we achieved further improvement over the original best performance, raising the overall score compared to the baseline by 1.13%.

5.1 Dataset Division Study

Based on the experiment we conducted on the full dataset, further investigations could be undertaken to visualize the results of data augmentation across different dataset sizes. A similar achievement was made by Wei and Zou (2019), who proposed the use of the EDA method. They achieved comparable results by training on a randomly sampled half of the examples from the data and matched the performance of the full dataset. This may highlight the significance of data augmentation, especially when comparing the effectiveness of the augmented dataset derived from the randomly sampled half to the other half of the original training dataset.

After considering the performance of all augmentation methods, we have ultimately chosen back-translation with pivot languages that yielded the highest scores: German and French. The process is illustrated in Figure 5.1. All data and their corresponding augmented versions are sampled randomly. Only half of the dataset, along with its augmented data, is utilized at this stage.

In Table 5.1, '0.5' refers to half of the dataset, while '0.5+0.5' denotes half of the original data combined with its corresponding back-translated data. From the table, the baseline

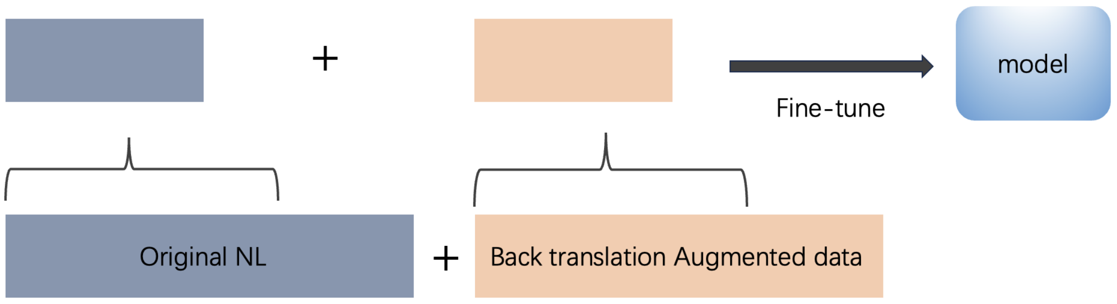

Figure 5.1: Augmentation on a random half of the dataset.

Model	**CodeBLEU**	**BLEU**
Baseline (0.5)	0.3611	0.2985
French (0.5+0.5)	0.3586	0.2934
German (0.5+0.5)	0.3531	0.2885
Baseline	0.3799	0.3203

Table 5.1: Experimental results for half of the data

for this model is about 1.7% higher than the one that used only half of the dataset. This indicates that the approximately, 4500 lines of the baseline data we used only resulted in minimal improvement for the model.

Another observation concerns the result of back-translation on half of the data. From the table, we note that both back-translation augmentations negatively impact the code model's performance. This is contrary to the results we obtained using the full dataset. This unexpected outcome might have several potential reasons. One possibility is that the randomly sampled half of the data differs in qualities compared to the other half, raising questions about whether it is cleaner or more informative.

5.2 Scale Issue of Back-Translation

By using a limited range of data and comparing it to the back-translation augmentation, we have observed that back-translation can negatively impact the code model's performance under certain conditions. Based on this observation, we believe that, the model's performance might not consistently improve as the volume of augmented back-translation data increases. This observation raises a question: do we need to use all the back-translation augmented data in text-to-code tasks?

Hence, we intend to validate these insights with an example. We will continue using back-translation with the two best pivot languages, German and French. We plan to increase the back-translation augmentation from 50% to 200% (where 200% represents a combination of both German and French). For the initial 100% of the augmented data, we aim to prioritize German due to its better performance and will introduce French in the range of 100% to 200%.

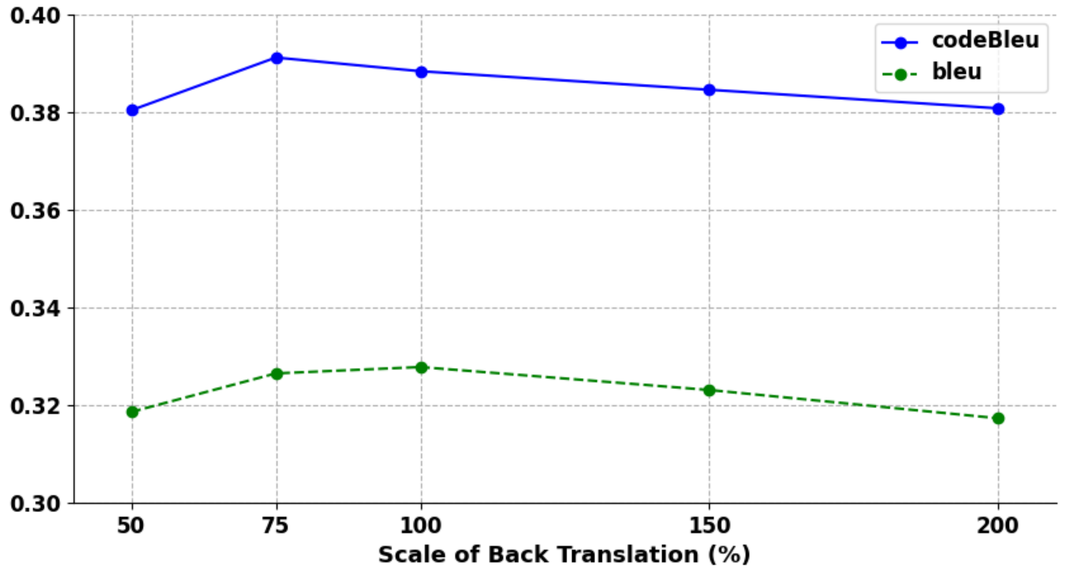

Figure 5.2: Scores varies with back-translation data scale

The results for the various ranges of the back-translation dataset are illustrated in Figure 5.2. This figure reveals that both the BLEU and CodeBLEU scores follow a parabolic trend. The peak for the CodeBLEU score occurs at 75% of the back-translation scale in the German dataset, indicating a performance superior to that of using the entire German dataset (100%). This observation suggests that more data is not always better; selecting a smaller, but potentially higher quality, augmented dataset might yield better performance.

Dataset	**CodeBLEU (%)**	**Δ (%)**	**BLEU (%)**
Baseline	37.99	-	32.03
50%	38.04	+0.05	31.86
75%	**39.12**	**+1.13**	32.65
100%	38.84	+0.85	**32.78**
150%	38.46	+0.47	32.31
200%	38.08	+0.09	31.73

Table 5.2: Experimental results for various ranges of the back-translation dataset.

More precise results compared to the baseline are shown in Table 5.2. It reveals that for all the back-translation scales we tested, the performance in terms of CodeBLEU score is better than the performance of the baseline. The highest performance is achieved with 75% of back-translation using pure German as the pivot language. This results in a CodeBLEU score of 39.12%, which is an improvement of 0.28% compared to the best performance augmentation score and 1.13% compared to the baseline.

Although this improvement might seem insignificant, it is noteworthy when compared to the results discussed earlier, where an additional 4500 lines in the baseline model resulted in only a slight 1.7% improvement when comparing the full dataset and half dataset. Therefore, the improvement achieved by back-translation at the right scale is significant.

Finally, the varying learning curves of different back-translation augmented datasets likely relate to the quality of the back-translation augmented data. The curves suggest that using fewer data can sometimes be more effective. Although back-translation is among the best-performing data augmentation methods, the NL in these instances might be less clear or more poorly described than in the original training set. Increasing the proportion of the augmented dataset and the original training dataset during fine-tuning could cause the model to over-rely on back-translated data. This might diminish the importance of the original training dataset, potentially leading to reduced model performance.

Chapter 6

Conclusion

6.1 Summary

The purpose of this thesis is to evaluate the effectiveness of data augmentation on text-to-code generation tasks. To achieve this, various data augmentation techniques were utilized to enhance the model's performance, resulting in a range of CodeBLEU scores. The methods employed are classified into three categories: word replacement, noise addition, and paraphrasing. Notably, the paraphrasing methods, specifically the Back-translation approach, outperformed others with a significant CodeBLEU score of 38.84% using German as the pivot language. However, although some techniques like paraphrasing and noise addition (word swap and deletion) showed promise with an appropriate selection of augmentations and probability replacements, others like word replacement and TF-IDF consistently underperformed or hovered around the baseline.

Overall, we obtained a 0.85% improvement compared to the baseline when employing back-translation. Although this result is not ideal, further analysis was conducted to determine whether the performance using half of the data is close to the performance of models fine-tuned using the full dataset. For this, half of the data was also subjected to the back-translation method to further verify its performance on smaller subsets. The results in Table 5.1 show that using only half of the data had only a 1.7% difference compared to using the full data, indicating more data is not necessarily better and highlighted the importance of less but good quality data. This suggests the need to reconsider the scale of back-translation for the full dataset to achieve higher performance. In addition, we illustrated BLEU and CodeBLEU scores as a function of the back-translation scale, showing a parabolic trend, where a scale of 75% resulted in a peak. This peak corresponds to a 39.12% CodeBLEU score, which is 0.28% higher than back-translation via German on the performance of the entire dataset and 1.13% higher than the baseline.

6.2 Evaluation

To evaluate the achievements of this experiment, it is crucial to first focus on the augmentation method. The effectiveness of various data augmentation techniques in text-to-code tasks was carefully examined. Several methods were evaluated, and the back-translation with German as the pivot language emerged as the most successful. Innovative techniques, such as GPT-3 paraphrasing, were also explored in this study, and they significantly contributed to the objective of comparing different augmentation methods. Further analysis of the scale of back-translation is a noteworthy point; a detailed analysis of the scale with the CodeBLEU score is worth investigating.

Although the improvement from the scaled back-translation, identified as our best-performing augmentation method, may seem marginal, it is important to notice that the baseline for this model is only about 1.7% higher than the one using only half of the dataset. This suggests that the approximately 4500 lines of baseline data utilized resulted in only minimal improvement for the model. Therefore, a 1.13% improvement achieved by back-translation should not be insignificant.

6.3 Future directions

In this experiment, our primary focus was on data augmentation for the text component of the text-to-code task. However, a further investigation could be conducted on the code part of this task. For the code part, modifications must be more strictly based on the programming language's structure; otherwise, it could lead to misleading results. For instance, we could create a dictionary of keywords that should not be replaced. Moreover, if we detect certain code structures, we could implement modifications that achieve the same outcomes, like transforming "a.sort()" into "sorted(a)". Method such as insertion of dead code, converting for loops to while loops or renaming variables can modify code without changing its output (You and Yim 2010). It is also feasible to integrate the augmentation techniques we employed in this experiment with those for code, enabling simultaneous augmentation of both text and code in text-to-code tasks.

Paraphrasing is one of the most effective augmentation methods in our experiments. Given the premise of this method, we could deploy a large code model that comprehends the code's semantics and provides corresponding textual explanations (Muennighoff et al. 2023). However, the NL part of XLCoST dataset we utilized was assembled by combining problem descriptions with related comments and follows a specific structure. To maintain such a structure when deploying sophisticated LLMs like GPT4, we can employ *few-shot* *prompts* (Brown et al. 2020a), in order to expose the model to similar examples.

Furthermore, since the structure of XLCoST's program level is formed by combining problem descriptions with related comments, each comment corresponds to a specific line of code. We can leverage this structure using techniques like *chain-of-thought prompting* (Wei et al. 2022). This technique primarily decomposes a complex question into multi-step, simpler questions. In this context, we can first have the LLM generate a problem description, followed by allowing the LLM to comment on each line of the provided code. Once these intermediate questions are solved, the answers can be combined, utilizing the reasoning abilities of LLM, to achieve a structure similar to the training set.

In Chapter 5, we observed that the size of the back-translation augmentation dataset is not always directly proportional to the performance of the fine-tuned model. The underlying reason might be related to the quality of the augmented data, such as its clarity and informativeness compared to the original training set. In further experiments, one should consider repeating this process by selecting multiple random subsets and averaging the results on the test set. This approach will help determine whether the observed issue is linked to the data quality, or if it is an inherent characteristic of the paraphrasing method in the text-to-code task. In the case of the latter, one can collect statistics on the augmented data, such as if the code is complete or if it is pruned (due to tokenization), which could reduce its clarity. Additionally, whether the NL descriptions are long or short might be indicative of their informativeness.

Bibliography

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Summarize and generate to back-translate: Unsupervised translation of programming languages. *arXiv preprint arXiv:2205.11116*, 2022.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yonatan Belinkov and Yonatan Bisk. Synthetic and natural noise both break neural machine translation. *arXiv preprint arXiv:1711.02173*, 2017.

Alexandre Berard, Ioan Calapodescu, and Claude Roux. Naver labs europe's systems for the wmt19 machine translation robustness task. *arXiv preprint arXiv:1907.06488*, 2019.

Chris M Bishop. Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1):108–116, 1995.

Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020a.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

Pinzhen Chen and Gerasimos Lampouras. Exploring data augmentation for code generation tasks. *arXiv preprint arXiv:2302.03499*, 2023.

Tri Dao, Albert Gu, Alexander Ratner, Virginia Smith, Chris De Sa, and Christopher Ré. A kernel theory of modern data augmentation. In *International conference on machine learning*, pages 1528–1537. PMLR, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A survey of data augmentation approaches for nlp. *arXiv preprint arXiv:2105.03075*, 2021.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. Coupling retrieval and meta-learning for context-dependent semantic parsing. *arXiv preprint arXiv:1906.07108*, 2019.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

Abhirut Gupta, Aditya Vavre, and Sunita Sarawagi. Training data augmentation for code-mixed translation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5760–5766, 2021.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Pei Liu, Xuemin Wang, Chao Xiang, and Weiye Meng. A survey of text data augmentation. In *2020 International Conference on Computer Communication and Network Security (CCNS)*, pages 191–195. IEEE, 2020.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38 (11):39–41, 1995.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Danish Pruthi, Bhuwan Dhingra, and Zachary C Lipton. Combating adversarial misspellings with robust word recognition. *arXiv preprint arXiv:1905.11268*, 2019.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*, 2016.

Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1017–1024, 2011.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proc. NIPS*, Montreal, CA, 2014. URL http://arxiv.org/abs/1409.3215.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Jason Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

Qizhe Xie, Zihang Dai, Eduard Hovy, Thang Luong, and Quoc Le. Unsupervised data augmentation for consistency training. *Advances in neural information processing systems*, 33:6256–6268, 2020.

Yiben Yang, Chaitanya Malaviya, Jared Fernandez, Swabha Swayamdipta, Ronan Le Bras, Ji-Ping Wang, Chandra Bhagavatula, Yejin Choi, and Doug Downey. Generative data augmentation for commonsense reasoning. *arXiv preprint arXiv:2004.11546*, 2020.

Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

Shiwen Yu, Ting Wang, and Ji Wang. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, 2023.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.